PyBDM Documentation

Release 0.1.0

Szymon Talaga

Sep 22, 2019

CONTENTS

1 Supported versions **3**

2 Usage **5**
- 2.1 Binary sequences (1D) . . . 5
- 2.2 Binary matrices (2D) . . . 5
- 2.3 Non-binary sequences (1D) . . . 6
- 2.4 Parallel processing . . . 6
- 2.5 Perturbation analysis . . . 7
- 2.6 Boundary conditions . . . 7
- 2.7 Normalized BDM . . . 8
- 2.8 Global options . . . 8
- 2.9 Advanced usage . . . 9

3 Theory & Design **11**
- 3.1 Algorithmic information theory . . . 11
- 3.2 Block Decomposition Method . . . 12
- 3.3 Implementation design . . . 13
- 3.4 Perturbation analysis . . . 14
- 3.5 References . . . 14

4 Contributing **15**
- 4.1 Types of Contributions . . . 15
- 4.2 Get Started! . . . 16
- 4.3 Pull Request Guidelines . . . 16
- 4.4 Tips . . . 16

5 Authors **17**
- 5.1 Core Team (AlgoDyn Development Team) . . . 17
- 5.2 Maintainer . . . 17
- 5.3 Contributors . . . 17

6 History **19**
- 6.1 0.1.0 (2019-09-22) . . . 19

7 Roadmap **21**
- 7.1 Next major release . . . 21
- 7.2 Distant goals . . . 21

8 pybdm **23**
- 8.1 pybdm package . . . 23

9 PyBDM: Python interface to the *Block Decomposition Method* **39**
9.1 Installation 39
9.2 Usage 40
9.3 Feedback 42

Bibliography **43**

Python Module Index **45**

Index **47**

Standard installation (stable):

```
pip install pybdm
```

Development version installation:

```
pip install git+https://github.com/sztal/pybdm.git
```

Local development:

```
git clone https://github.com/sztal/pybdm
cd pybdm
pip install --editable .
```

CHAPTER

ONE

SUPPORTED VERSIONS

Python3.5+ is supported. Tests are run against Linux, but Windows and OSX should work as well.

CHAPTER

TWO

USAGE

The BDM is implemented using the object-oriented approach and expects input represented as Numpy arrays of integer type.

> `BDM` objects operate exclusively on **integer arrays**. Hence, any alphabet must be first mapped to a set of integers ranging from `0` to `k`. Currently only standard `numpy` arrays are accepted. However, in general it is possible to conceive of a BDM variant optimized for sparse array. We plan provide in the releas.

Detailed description of the design of our implementation of BDM can be found in *Theory & Design*.

2.1 Binary sequences (1D)

```
import numpy as np
from pybdm import BDM

# Create a dataset (must be of integer type)
X = np.ones((100,), dtype=int)

# Initialize BDM object
# ndim argument specifies dimensionality of BDM
bdm = BDM(ndim=1)

# Compute BDM
bdm.bdm(X)

# BDM objects may also compute standard Shannon entropy in base 2
bdm.ent(X)
```

2.2 Binary matrices (2D)

```
import numpy as np
from pybdm import BDM

# Create a dataset (must be of integer type)
X = np.ones((100, 100), dtype=int)

# Initialize BDM object
bdm = BDM(ndim=2)

# Compute BDM
```

(continues on next page)

(continued from previous page)

```
bdm.bdm(X)

# BDM objects may also compute standard Shannon entropy in base 2
bdm.ent(X)
```

2.3 Non-binary sequences (1D)

```
import numpy as np
from pybdm import BDM

# Create a dataset (4 discrete symbols)
np.random.seed(303)
X = np.random.randint(0, 4, (100,))

# Initialize BDM object with 4-symbols alphabet
bdm = BDM(ndim=1, nsymbols=4)

# Compute BDM
bdm.bdm(X)
```

2.4 Parallel processing

PyBDM was designed with parallel processing in mind. Using modern packages for parallelization such as joblib makes it really easy to compute BDM for massive objects.

In this example we will slice a 1000x1000 dataset into 200x200 pieces compute so-called counter objects (final BDM computation operates on such objects) in parallel in 4 independent processes, and aggregate the results into a single BDM approximation of the algorithmic complexity of the dataset.

> Remember that data has to be sliced correctly during parallelization in order to ensure fully correct BDM computations. That is, all slices except lower and right boundaries have to be decomposable without any boundary leftovers by the selected decomposition algorithm.

```
import numpy as np
from joblib import Parallel, delayed
from pybdm import BDM
from pybdm.utils import decompose_dataset

# Create a dataset (must be of integer type)
X = np.ones((1000, 1000), dtype=int)

# Initialize BDM object
bdm = BDM(ndim=2)

# Compute counter objects in parallel
counters = Parallel(n_jobs=4) \
    (delayed(bdm.decompose_and_count)(d) for d in decompose_dataset(X, (200, 200)))

# Compute BDM
bdm.compute_bdm(*counters)
```

2.5 Perturbation analysis

Besides the main *Block Decomposition Method* implementation *PyBDM* provides also an efficient algorithm for perturbation analysis based on *BDM* (or standard Shannon entropy).

A perturbation experiment studies change of *BDM* / entropy under changes applied to the underlying dataset. This is the main tool for detecting parts of a system having some causal significance as opposed to noise parts.

Parts which after yield negative contribution to the overall complexity after change are likely to be important for the system, since their removal make it more noisy. On the other hand parts that yield positive contribution to the overall complexity after change are likely to be noise since they extend the system's description length.

```
import numpy as np
from pybdm import BDM
from pybdm.algorithms import PerturbationExperiment

# Create a dataset (must be of integer type)
X = np.ones((100, 100), dtype=int)

# Initialize BDM object
bdm = BDM(ndim=2)

# Initialize perturbation experiment object
# (may be run for both bdm or entropy)
perturbation = PerturbationExperiment(bdm, X, metric='bdm')

# Compute BDM change for all data points
delta_bdm = perturbation.run()

# Compute BDM change for selected data points and keep the changes while running
# One array provide indices of elements that are to be change.
idx = np.array([[0, 0], [10, 10]], dtype=int)
# Another array provide new values to assign.
# Negative values mean that new values will be selected
# randomly from the set of other possible values from the alphabet.
values = np.array([-1, -1], dtype=int)
delta_bdm = perturbation.run(idx, values, keep_changes=True)

# Here is an example applied to an adjacency matrix
# (only 1's are perturbed and switched to 0's)
# (so perturbations correspond to edge deletions)
X = np.random.randint(0, 2, (100, 100))
# Indices of nonzero entries in the matrix
idx = np.argwhere(X)
# PerturbationExperiment can be instantiated without passing data
pe = PerturbationExperiment(bdm, metric='bdm')
# data can be added later
pe.set_data(X)
# Run experiment and perturb edges
# No values argument is passed so perturbations automatically switch
# values to other values from the alphabet (in this case 1 --> 0)
delta_bdm = pe.run(idx)
```

2.6 Boundary conditions

Different boundary conditions (see *Theory & Design*) are implemented by `partitions` classes.

```
from pybdm import BDM
from pybdm import PartitionIgnore, PartitionRecursive, PartitionCorrelated

bdm_ignore = BDM(ndim=1, partition=PartitionIgnore)
# This is default so it is equivalent to
bdm_ignore = BDM(ndim=1)

bdm_recurisve = BDM(ndim=1, partition=PartitionRecursive, min_length=2)
# Minimum size is specified as length, since only symmetric slices
# are accepted in the case of multidimensional objects.

bdm_correlated = BDM(ndim=1, partition=PartitionCorrelated)
# Step-size defaults to 1, so this is equivalent to
bdm_correlated = BDM(ndim=1, partition=PartitionCorrelated, shift=1)
```

2.7 Normalized BDM

It is also possible to compute normalized BDM and block entropy values which are always bounded in the [0, 1] interval.

```
import numpy as np
for pybdm import BDM

# Minimally complex data
X = np.ones((100,), dtype=int)

bdm = BDM(ndim=1)

# Normalized BDM (equals zero in this case)
bdm.nbdm(X)
# Equivalent call
bdm.bdm(X, normalized=True)

# Normalized entropy (equals zero in this case)
bdm.nent(X)
# Equivalent call
bdm.ent(X, normalized=True)
```

2.8 Global options

Some parts of the behavior of the package can be configured globally via package-level options.

Options are documented in the module docstring for *pybdm.options*.

```
from pybdm import options
# Get a copy of the current options dict
options.get()
# Get the current value of an option
options.get('raise_if_zero')
# Set and option
options.set(raise_if_zero=False)
```

2.9 Advanced usage

Advanced usage and details can be found in the *pybdm* module documentation.

CHAPTER

THREE

THEORY & DESIGN

The Block Decomposition Method (BDM) approximates algorithmic complexity of a dataset of arbitrary size, that is, the length of the shortest computer program that generates it. This is not trivial as algorithmic complexity is not a computable quantity in the general case and estimation of algorithmic complexity of a dataset can be very useful as it points to mechanistic connections between elements of a system, even such that do not yield any regular statistical patterns that can be captured with more traditional tools based on probability theory and information theory.

Currently 1D and 2D binary arrays are supported, as well as 1D arrays with 4, 5, 6 and 9 discrete symbols.

BDM and the necessary parts of the algorithmic information theory it is based on are described in [STZDG14] and [ZHOK+18].

3.1 Algorithmic information theory

Here we give a super brief and simplified overview of the basic notions of algorithmic information theory, which we will need to describe the implementation of the package.

Algorithmic / Kolmogorv complexity (also called K-complexity) is defined formally as follows:

$$K_U = \min\{|p|, T(p) = s\}$$

where U is a universal Turing machine, p is a program, $|p|$ is the lenght of the program, s is a string $U(p) = s$ denotes the fact that the program p executed on the universal Turing machine U outputs s.

The problem with Kolmogorov complexity is the fact that it is not computable in the general case due to fundamental limits of computations that arise from the halting problem (impossibility to determine whether any given program will ever halt without actually running this program, possibly for infinite time).

It is also possible to consider the notion of algorithmic probability, which corresponds to a chance that a randomly selected program will output s when run through U. It is defined as follows:

$$m_U(s) = \sum_{p:U(p)=s} 1/2^{|p|}$$

Algorithmic probability is important because it is related directly to algorithmic complexity via the following law:

$$K_U(s) = -\log_2 m_U(s) + O(1)$$

In other words, if there are many long programs that generate a dataset, then there has to be also a shorter one. The arbitrary constant $O(1)$ is dependent on the choice of a programming language.

Unfortunately, algorithmic probability is also uncomputable for the same reasons as Kolmogorov complexity. However, it can be approximated in a very straightforward fashion, since it is possible to explore a vast space of Turing machines of a given type (i.e. fixed numbers of symbols and states) and count how many of them produce a given

output and then divide by the total number of machines that halt. Details of how this can be done can be found in [STZDG14]. Thus, when exploring machines with n symbols and m states algorithmic probability of a string s can be approximated as follows:

$$D(n,m)(s) = \frac{|\{T \in (n,m) : T \text{ outputs } s\}|}{|\{T \in (n,m) : T \text{ halts }\}|}$$

Based on that we can approximate Kolmogorov complexity (via the so-called Coding Theorem Method) as follows:

$$CTM(n,m)(s) = -\log_2 D(n,m)(s)$$

This is the basic result that is used to define Block Decomposition Method.

3.2 Block Decomposition Method

The problem with CTM is that, although theoretically computable, it is still extremely expensive in terms of computation time, since it depends on exploration of vast spaces of possible Turing machines that may span bilions or even thousands of bilions of instances. This problem is what Block Decomposition Method (BDM) tries to address [ZHOK+18].

The idea is to first precompute CTM values for all possible small objects of a given type (e.g. all binary strings of up to 12 digits or all possible square binary matrices up to 4x4) and store them in an efficient lookup table. Then any arbitrarily large object can be decomposed into smaller slices of appropriate sizes for which CTM values can be looked up very fast. Finally, the CTM valeus for slices can be aggregated back to a global estimate of Kolmogorov complexity for the entire object. The proper aggregation rule is defined via the following BDM formula:

$$BDM(n,m)(s) = \sum_i CTM(n,m)(s_i) + \log_2(n_i)$$

where i indexes the set of all unique slices (i.e. CTM values are taken only once for each unique slice) and n_i correspond to the slices' numbers of occurences.

Available CTM datasets are listed in `pybdm.ctmdata`.

3.2.1 Boundary conditions

A technical problem that arises in the context of BDM is what should be done if a dataset can not be sliced into parts of the same extact shape? There are at least three solutions:

1. **Ignore.** Malformed parts can be just ignored.
2. **Recursive.** Slice malformed parts into smaller pieces (down to some minimum size) and lookup CTM values for those smaller pieces.
3. **Correlated.** Use sliding window instead of slicing. This way all slices will be of the proper shape, at least if the window is moved by one element at every step.

Let us show how this works with a simple example. Let us consider a 5-by-5 matrix:

1	2	3	4	5
6	7	8	9	10
11	12	13	14	15
16	17	18	19	20
21	22	23	24	25

If the ignore boundary condition was used, only one 3-by-3 matrix would be carved out of it:

1	2	3
6	7	8
11	12	13

If the recursive condition (with 2-by-2 as the minimum size) was used, we would get the following slices:

1	2	3
6	7	8
11	12	13

4	5
9	10

16	17
21	22

18	18
23	24

If the correlated condition (with step-size of 1) was used, we would get nine slices like:

1	2	3
6	7	8
11	12	13

but each subsequent slice would be moved by one to the left or to the bottom until its rightmost column or lowest row contain the values on the boundary of the original matrix.

The condition can yield different results for small objects, but are consistent asymptotically in the limit of large object sizes. Detailed discussion of boundary conditions in BDM can be found in [ZHOK+18].

3.2.2 Normalized BDM

It is also possible to define normalized BDM. First let us note that for any object of arbitrary size it is possible to construct analogous objects wit lowest and highest possible BDM values.

- **Least complex object.** This case is trivial. It is enough to consider an object filled with only one symbol (e.g. a binary string of only zeros).
- **Most complex object.** The maximum BDM value is given by an object which when decomposed (by a given decomposition algorithm) yields slices that cover the highest CTM values and are repeated only after all possible slices of a given shape have been used once.

3.3 Implementation design

The implementation uses the OOP pattern and follow the *split-apply-comine* methodology. There are two main classes:

pybdm.bdm.BDM Instances of this class contain pointers to appropriate precomputed CTM datasets. They configured by two main attributes: dimensionality of target objects (*ndim*) and number of symbols used (*nsymbols*). The class implements BDM in three stages. The first one is decomposition which relies on a particular partition algorithm object (below). The second one is lookup (CTM values for slices are looked up). The third one is aggregation, in which CTM values for slices are combined according to the BDM formula. This stage-wise implementation makes it easy to extend the package for instance with new partition algorithms and also makes it very easy to parallelize or distribute the entire process.

pybdm.partitions Decomposition stage is implemented by partition classes. They are instantiated with attributes describind desired shape of slices, step-sizes (*shift*) in correlated decomposition, minimum size in recursive decomposition etc. Partition objects are used by BDM objects during the decomposition stage.

See *Usage* for practical examples.

3.3.1 Missing CTM values

In some cases (especially for alphabets with more than 2 symbols) CTM values for particular slices may not be available. They are imputed with the maximum CTM value for slices of a given shape + 1 bit. This is justified because the exploration of the spaces of Turing machines is done in a way that ensures that missed value can be only larger than those that were computed.

By default BDM objects send warnings when this happens. However, this may be turned off:

```
# For a particular BDM instance
bdm = BDM(ndim=1, warn_if_missing_ctm=False)

# Or globally for all BDM instances
pybdm.options.set(warn_if_missing_ctm=False)
```

3.3.2 Block entropy

pybdm.bdm.BDM class implements also *ent* method for computing block entropy, which is useful for comparisons between algorithmic complexity and entropy as another measure of description length. Block entropy is just the entropy computed over the distribution of slices as produced by the partition algorithm.

3.4 Perturbation analysis

pybdm provides also an efficient algorithm for perturbation analysis. The goal of perturbation analysis is to study changes in complexity of a system under small changes. This makes it possible to identify parts that drive it towards noise (high complexity) or determinism / structure (low complexity).

For instance, it may be of interest to examine changes of complexity of an adjacency matrix when particular edges are destroyed (ones switched to zeros). Some practical applications of such analysis can be found in [ZKZT19].

3.5 References

CHAPTER

FOUR

CONTRIBUTING

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

You can contribute in many ways:

4.1 Types of Contributions

4.1.1 Report Bugs

Report bugs at https://github.com/sztal/pybdm/issues.

If you are reporting a bug, please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

4.1.2 Fix Bugs

Look through the GitHub issues for bugs. Anything tagged with "bug" is open to whoever wants to implement it.

4.1.3 Implement Features

Look through the GitHub issues for features. Anything tagged with "feature" is open to whoever wants to implement it.

4.1.4 Write Documentation

PyBDM could always use more documentation, whether as part of the official PyBDM docs, in docstrings, or even on the web in blog posts, articles, and such.

4.1.5 Submit Feedback

The best way to send feedback is to file an issue at https://github.com/sztal/pybdm/issues.

If you are proposing a feature:

- Explain in detail how it would work.

- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that contributions are welcome :)

4.2 Get Started!

Ready to contribute? Here's how to set up *bdm* for local development.

1. Fork the *pybdm* repo on GitHub.
2. Clone your fork locally:

```
$ git clone git@github.com:your_name_here/pybdm.git
```

3. Create a branch for local development:

```
$ git checkout -b name-of-your-bugfix-or-feature
```

Now you can make your changes locally.

4. When you're done making changes, check that your changes pass style and unit tests, including testing other Python versions with tox:

```
$ tox
```

To get tox, just pip install it.

5. Commit your changes and push your branch to GitHub:

```
$ git add .
$ git commit -m "Your detailed description of your changes."
$ git push origin name-of-your-bugfix-or-feature
```

6. Submit a pull request through the GitHub website.

4.3 Pull Request Guidelines

Before you submit a pull request, check that it meets these guidelines:

1. The pull request should include tests.
2. If the pull request adds functionality, the docs should be updated. Put your new functionality into a function with a docstring, and add the feature to the list in README.rst.
3. The pull request should work for Python 3.5, 3.6 and 3.7 and for PyPy. Check https://travis-ci.org/sztal/pybdm under pull requests for active pull requests or run the `tox` command and make sure that the tests pass for all supported Python versions.

4.4 Tips

To run a subset of tests:

```
$ py.test test/test_bdm.py
```

CHAPTER

FIVE

AUTHORS

5.1 Core Team (AlgoDyn Development Team)

- Szymon Talaga <stalaga@protonmail.com>
- Kostas Tsampourakis <kostas.tsampourakis@gmail.com>

5.2 Maintainer

- Szymon Talaga <stalaga@protonmail.com>

5.3 Contributors

- Paweł Weroński <pawel.m.weronski@gmail.com>

CHAPTER

SIX

HISTORY

6.1 0.1.0 (2019-09-22)

- First release on PyPI.

CHAPTER

SEVEN

ROADMAP

7.1 Next major release

- Support for sparse arrays
- Perturbation experiment for growing/shrinking systems
- Implement Bayesian framework for approximating probability of a stochastic generating source
- Add a partition algorithm with the periodic boundary condition
- Use integer-based conding of dataset blocks (to lower memory-footprint). This will be done only if it will be possible to use integer coding without significantly negative impact on the performance.
- Configure automatic tests for OSX and Windows.

7.2 Distant goals

- Reimplement core algorithms and classes in C++ and access them via cython

CHAPTER

EIGHT

PYBDM

8.1 pybdm package

8.1.1 Subpackages

pybdm.ctmdata package

Module contents

Resources submodule with reference dataset containing precomputed approximated algorithmic complexity values for simple objects based on *Coding Theorem Method* (see *Theory & Design*).

All datasets' names use the following naming scheme: `ctm-bX-dY`.

Datasets

`ctm-b2-d12.pkl` Binary strings of length from 1 to 12.

`ctm-b4-d12.pkl` 4-symbols strings of length from 1 to 12.

`ctm-b5-d12.pkl` 5-symbols strings of length from 1 to 12.

`ctm-b6-d12.pkl` 6-symbols strings of length from 1 to 12.

`ctm-b9-d12.pkl` 9-symbols strings of length from 1 to 12.

`ctm-b2-d4x4.pkl` Square binary matrices of width from 1 to 4.

8.1.2 Submodules

8.1.3 pybdm.algorithms module

Algorithms based on `BDM` objects.

class `pybdm.algorithms.`**PerturbationExperiment**(*bdm*, *X=None*, *metric='bdm'*)

Bases: `object`

Perturbation experiment class.

Perturbation experiment studies change of BDM / entropy under changes applied to the underlying dataset. This is the main tool for detecting parts of a system having some causal significance as opposed to noise parts.

Parts which when perturbed yield negative contribution to the overall complexity after change are likely to be important for the system, since their removal make it more noisy. On the other hand parts that yield positive contribution to the overall complexity after change are likely to be noise since they elongate the system's description length.

bdm
BDM object. It has to be configured properly to handle the dataset that is to be studied.

Type *BDM*

X
Dataset for perturbation analysis. May be set later.

Type array_like (optional)

metric
Which metric to use for perturbing.

Type {'bdm', 'ent'}

See also:

pybdm.bdm.BDM BDM computations

Examples

```
>>> import numpy as np
>>> from pybdm import BDM, PerturbationExperiment
>>> X = np.random.randint(0, 2, (100, 100))
>>> bdm = BDM(ndim=2)
>>> pe = PerturbationExperiment(bdm, metric='bdm')
>>> pe.set_data(X)
>>> idx = np.argwhere(X) # Perturb only ones (1 --> 0)
>>> delta_bdm = pe.run(idx)
>>> len(delta_bdm) == idx.shape[0]
True
```

More examples can be found in *Usage*.

property ndim
Data number of axes getter.

perturb(*idx*, *value=-1*, *keep_changes=False*)
Perturb element of the dataset.

Parameters

- **idx** (*tuple*) – Index tuple of an element.
- **value** (*int or callable or None*) – Value to assign. If negative then new value is randomly selected from the set of other possible values. For binary data this is just a bit flip and no random numbers generation is involved in the process.
- **keep_changes** (*bool*) – If `True` then changes in the dataset are persistent, so each perturbation step depends on the previous ones.

Returns BDM value change.

Return type float

Examples

```
>>> from pybdm import BDM
>>> bdm = BDM(ndim=1)
>>> X = np.ones((30, ), dtype=int)
>>> perturbation = PerturbationExperiment(bdm, X)
>>> perturbation.perturb((10, ), -1)
26.91763012739709
```

run(*idx=None*, *values=None*, *keep_changes=False*)
Run perturbation experiment.

Parameters

- **idx** (*array_like or None*) – *Numpy* integer array providing indexes (in rows) of elements to perturb. If `None` then all elements are perturbed.
- **values** (*array_like or None*) – Value to assign during perturbation. Negative values correspond to changing value to other randomly selected symbols from the alphabet. If `None` then all values are assigned this way. If set then its dimensions must agree with the dimensions of `idx` (they are horizontally stacked).
- **keep_changes** (*bool*) – If `True` then changes in the dataset are persistent, so each perturbation step depends on the previous ones.

Returns 1D float array with perturbation values.

Return type array_like

Examples

```
>>> from pybdm import BDM
>>> bdm = BDM(ndim=1)
>>> X = np.ones((30, ), dtype=int)
>>> perturbation = PerturbationExperiment(bdm, X)
>>> changes = np.array([10, 20])
>>> perturbation.run(changes)
array([26.91763013, 27.34823681])
```

set_data(*X*)
Set dataset for the perturbation experiment.

Parameters **X** (*array_like*) – Dataset to perturb.

property shape
Data shape getter.

property size
Data size getter.

8.1.4 pybdm.bdm module

Block Decomposition Method

BDM class provides a top-level interface for configuring an instance of a block decomposition method as well as running actual computations approximating algorithmic complexity of given datasets.

Configuration step is necessary for specifying dimensionality of allowed datasets, reference CTM data as well as boundary conditions for block decomposition etc. This is why BDM is implemented in an object-oriented fashion, so an instance can be first configured properly and then it exposes a public method `BDM.bdm()` for computing approximated complexity via BDM.

class pybdm.bdm.**BDM**(*ndim*, *nsymbols=2*, *shape=None*, *partition=<class 'pybdm.partitions.PartitionIgnore'>*, *ctmname=None*, *warn_if_missing_ctm=True*, *raise_if_zero=True*, ***kwds*)

Bases: `object`

Block decomposition method.

ndim
Number of dimensions of target dataset objects. Positive integer.

Type int

nsymbols
Number of symbols in the alphabet.

Type int

partition
Partition algorithm class object. The class is called with the *shape* attribute determined automatically if not passed and other attributes passed via `**kwds`.

Type Partition class

ctmname
Name of the CTM dataset. If `None` then a CTM dataset is selected automatically based on *ndim* and *nsymbols*.

Type str

warn_if_missing_ctm
Should `BDMRuntimeWarning` be sent in the case there is missing CTM value. Some CTM values may be missing for larger alphabets as it is computationally infeasible to explore entire parts space. Missing CTM values are imputed with mean CTM complexities over all parts of a given shape. This can be also disabled globally with the global option of the same name, i.e. `pybdm.options.set(warn_if_missing_ctm=False)`.

Type bool

raise_if_zero
Should error be raised if BDM value is zero. Zero value indicates that a dataset could have incorrect dimensions. This can be also disabled globally with the global option of the same name, i.e. `pybdm.options.set(raise_if_zero=False)`.

Type bool

Notes

Block decomposition method in **PyBDM** is implemented in an object-oriented fashion. This design choice was dictated by the fact that BDM can not be applied willy-nilly to any dataset, but has to be configured for a particular type of data (e.g. binary matrices). Hence, it is more convenient to first configure and instatiate a particular instance of BDM and the apply it freely to data instead of passing a lot of arguments at every call.

BDM has also natural structure corresponding to the so-called *split-apply-combine* strategy in data analysis. First, a large dataset it decomposed into smaller block for which precomputed CTM values can be efficiently looked up. Then CTM values for slices are aggregated in a theory-informed way into a global approximation of complexity of the full dataset. Thus, BDM computations naturally decomposes into four stages:

1. **Partition (decomposition) stage.** First a dataset is decomposed into block. This is done by the *decompose()* method. The method itself is dependent on the *partition* attribute which points to a *pybdm.partitions* object, which implements and configures a particular variant of the decomposition algorithm. Detailed description of the available algorithms can be found in *Theory & Design*.
2. **Lookup stage.** At this stage CTM values for blocks are looked up. This is when the CTM reference dataset is used. It is implemented in the :py:meth‘lookup‘ method.
3. **Count stage.** Unique dataset blocks are counted and arranged in an efficient data structure together with their CTM values.
4. **Aggregate stage.** Final BDM value is computed based on block counter data structure.

See also:

pybdm.ctmdata available CTM datasets

pybdm.partitions available partition and boundary condition classes

bdm(*X*, *normalized=False*, *check_data=True*)

Approximate complexity of a dataset with BDM.

Parameters

- **X** (*array_like*) – Dataset representation as a `numpy.ndarray`. Number of axes must agree with the *ndim* attribute.
- **normalized** (*bool*) – Should BDM be normalized to be in the [0, 1] range.
- **check_data** (*bool*) – Should data format be checked. May be disabled to gain some speed when calling multiple times.

Returns Approximate algorithmic complexity.

Return type float

Raises

- **TypeError** – If *X* is not an integer array and *check_data=True*.
- **ValueError** – If *X* has more than *nsymbols* unique values and *check_data=True*.
- **ValueError** – If *X* has symbols outside of the `0` to *nsymbols-1* range and *check_data=True*.
- **ValueError** – If computed BDM value is 0 and *raise_if_zero* is `True`.

Notes

Detailed description can be found in *Theory & Design*.

Examples

```
>>> import numpy as np
>>> bdm = BDM(ndim=2)
>>> bdm.bdm(np.ones((12, 12), dtype=int))
25.176631293734488
```

compute_bdm(**counters*)

Approximate Kolmogorov complexity based on the BDM formula.

Parameters ***counters** – Counter objects grouping object keys and occurences.

Returns Approximate algorithmic complexity.

Return type float

Notes

Detailed description can be found in *Theory & Design*.

Examples

```
>>> from collections import Counter
>>> bdm = BDM(ndim=1)
>>> c1 = Counter([('111111111111', 1.95207842085224e-08)])
>>> c2 = Counter([('111111111111', 1.95207842085224e-08)])
>>> bdm.compute_bdm(c1, c2)
1.000000019520784
```

compute_ent(**counters*)

Compute block entropy from a counter obejct.

Parameters ***counters** – Counter objects grouping object keys and occurences.

Returns Block entropy in base 2.

Return type float

Examples

```
>>> from collections import Counter
>>> bdm = BDM(ndim=1)
>>> c1 = Counter([('111111111111', 1.95207842085224e-08)])
>>> c2 = Counter([('000000000000', 1.95207842085224e-08)])
>>> bdm.compute_ent(c1, c2)
1.0
```

count(*ctms*)

Count unique blocks.

Parameters **ctms** (*sequence of 2-tuples*) – Ordered 1D sequence of string keys and CTM values.

Returns Set of unique blocks with their CTM values and numbers of occurences.

Return type Counter

Examples

```
>>> bdm = BDM(ndim=1)
>>> data = np.ones((24, ), dtype=int)
>>> parts = bdm.decompose(data)
>>> ctms = bdm.lookup(parts)
>>> bdm.count(ctms)
Counter({('111111111111', 25.610413747641715): 2})
```

decompose(*X*)

Decompose a dataset into blocks.

Parameters **x** (*array_like*) – Dataset of arbitrary dimensionality represented as a *Numpy* array.

Yields *array_like* – Dataset blocks.

Raises **AttributeError** – If blocks' *shape* and dataset's shape have different numbers of axes.

Acknowledgments

Special thanks go to Paweł Weroński for the help with the design of the non-recursive *partition* algorithm.

Examples

```
>>> bdm = BDM(ndim=2, shape=(2, 2))
>>> [ x for x in bdm.decompose(np.ones((4, 3), dtype=int)) ]
[array([[1, 1],
       [1, 1]]), array([[1, 1],
       [1, 1]])]
```

decompose_and_count(*X*, *lookup_ctm=True*)

Decompose and count blocks.

Parameters

- **X** (*array_like*) – Dataset representation as a `numpy.ndarray`. Number of axes must agree with the *ndim* attribute.
- **lookup_ctm** (*bool*) – Should CTM values be looked up.

Returns Lookup table mapping 2-tuples with string keys and CTM values to numbers of occurences.

Return type collections.Counter

Notes

This is equivalent to calling *decompose()*, *lookup()* and *count()*.

Examples

```
>>> import numpy as np
>>> bdm = BDM(ndim=1)
>>> bdm.decompose_and_count(np.ones((12, ), dtype=int))
Counter({('111111111111', 25.610413747641715): 1})
```

ent(*X*, *normalized=False*, *check_data=True*)

Block entropy of a dataset.

Parameters

- **X** (*array_like*) – Dataset representation as a `numpy.ndarray`. Number of axes must agree with the *ndim* attribute.
- **normalized** (*bool*) – Should entropy be normalized to be in the [0, 1] range.

- **check_data** (*bool*) – Should data format be checked. May be disabled to gain some speed when calling multiple times.

Returns Block entropy in base 2.

Return type float

Raises

- **TypeError** – If *X* is not an integer array and *check_data=True*.
- **ValueError** – If *X* has more than *nsymbols* unique values and *check_data=True*.
- **ValueError** – If *X* has symbols outside of the `0` to *nsymbols-1* range and *check_data=True*.

Examples

```
>>> import numpy as np
>>> bdm = BDM(ndim=2)
>>> bdm.ent(np.ones((12, 12), dtype=int))
0.0
```

lookup(*blocks*, *lookup_ctm=True*)

Lookup CTM values for blocks in a reference dataset.

Parameters

- **blocks** (*sequence*) – Ordered sequence of dataset parts.
- **lookup_ctm** (*bool*) – Should CTM values be looked up.

Yields *tuple* – 2-tuple with string representation of a dataset part and its CTM value.

Raises **KeyError** – If key of an object can not be found in the reference CTM lookup table.

Warns **BDMRuntimeWarning** – If `warn_if_missing_ctm=True` and there is no pre-computed CTM value for a part during the lookup stage. This can be always disabled with the global option of the same name.

Examples

```
>>> bdm = BDM(ndim=1)
>>> data = np.ones((12, ), dtype=int)
>>> parts = bdm.decompose(data)
>>> [ x for x in bdm.lookup(parts) ]
[('111111111111', 25.610413747641715)]
```

nbdm(*X*, ***kwds*)

Alias for normalized BDM

Other arguments are passed as keywords.

See also:

bdm() BDM method

nent (*X*, ***kwds*)
Alias for normalized block entropy.

Other arguments are passed as keywords.

See also:

ent() block entropy method

8.1.5 pybdm.encoding module

Encoding and decoding of arrays with fixed number of unique symbols.

While computing BDM dataset blocks have to be encoded into simple hashable objects such as strings or integers for efficient lookup of CTM values from reference datasets.

Currently string-based keys are used in CTM datasets. However, this may be changed to integer keys in the future in order to lower the memory footprint.

Integer encoding can be also used for easy generation of objects of fixed dimensionality as each such object using a fixed, finite alphabet of symbols can be uniquely mapped to an integer code.

pybdm.encoding.**array_from_string**(*x*, *shape*, *cast_to=<class 'int'>*)
Make array from string code.

Parameters

- **x** (*str*) – String code.
- **shape** (*tuple*) – Desired shape of the output array.
- **cast_to** (*type or None*) – Cast array to given type. No casting if `None`. Defaults to integer type.

Returns Array encoded in the string code.

Return type array_like

Examples

```
>>> array_from_string('1010', shape=(4,))
array([1, 0, 1, 0])
>>> array_from_string('1000', shape=(2, 2))
array([[1, 0],
       [0, 0]])
```

pybdm.encoding.**decode_array**(*code*, *shape*, *base=2*, ***kwds*)
Decode array of integer-symbols from a sequence code.

Parameters

- **code** (*int*) – Non-negative integer.
- **shape** (*tuple of ints*) – Expected array shape.
- **base** (*int*) – Encoding base.
- ****kwds** – Keyword arguments passed to `numpy.reshape()`.

Returns *Numpy* array.

Return type array_like

pybdm.encoding.**decode_sequence**(*code*, *base=2*, *min_length=None*)
Decode sequence from a sequence code.

Parameters

- **code** (*int*) – Non-negative integer.
- **base** (*int*) – Encoding base. Should be equal to the number of unique symbols in the alphabet.
- **min_length** (*int or None*) – Minimal number of represented bits. Use shortest representation if None.

Returns 1D *Numpy* array.

Return type array_like

Examples

```
>>> decode_sequence(4)
array([1, 0, 0])
```

pybdm.encoding.**encode_array**(*x*, *base=2*, ***kwds*)
Encode array of integer-symbols.

Parameters

- **x** (*(N, k) array_like*) – Array of integer symbols.
- **base** (*int*) – Encoding base.
- ****kwds** – Keyword arguments passed to numpy.ravel().

Returns Integer code of an array.

Return type int

pybdm.encoding.**encode_sequence**(*seq*, *base=2*)
Encode sequence of integer-symbols.

Parameters

- **seq** (*(N,) array_like*) – Sequence of integer symbols represented as 1D *Numpy* array.
- **base** (*int*) – Encoding base. Should be equal to the number of unique symbols in the alphabet.

Returns Integer code of a sequence.

Return type int

Raises

- **AttributeError** – If *seq* is not 1D.
- **TypeError** – If *seq* is not of integer type.
- **ValueError** – If *seq* contain values which are negative or beyond the size of the alphabet (encoding base).

Examples

```
>>> encode_sequence(np.array([1, 0, 0]))
4
```

pybdm.encoding.**normalize_array**(*X*)

Normalize array so symbols are consecutively mapped to 0, 1, 2, ...

Parameters **X** (*array_like*) – *Numpy* array of arbitrary dimensions.

Returns *Numpy* array of the same dimensions with mapped symbols.

Return type array_like

Examples

```
>>> X = np.array([1, 2, 3], dtype=int)
>>> normalize_array(X)
array([0, 1, 2])
>>> X = np.array([[1,2],[2,1]], dtype=int)
>>> normalize_array(X)
array([[0, 1],
       [1, 0]])
```

pybdm.encoding.**normalize_key**(*key*)

Normalize part key so symbols are consecutively mapped to 0, 1, 2, ...

Parameters **key** (*str*) – Part key as returned by *string_from_array()*.

Returns Normalized key with mapped symbols.

Return type str

Examples

```
>>> normalize_key('123')
'012'
>>> normalize_key('40524')
'01230'
```

pybdm.encoding.**string_from_array**(*arr*)

Encode an array as a string code.

Parameters **arr** (*(N, k) array_like*) – *Numpy* array.

Returns String code of an array.

Return type str

Examples

```
>>> string_from_array(np.array([1, 0, 0]))
'100'
>>> string_from_array(np.array([[1,0], [3,4]]))
'1034'
```

8.1.6 pybdm.exceptions module

PyBDM exception and warning classes.

exception pybdm.exceptions.**BDMConfigurationError**
Bases: AttributeError

General BDM configuration error.

exception pybdm.exceptions.**BDMRuntimeWarning**
Bases: RuntimeWarning

General BDM related runtime warning class.

exception pybdm.exceptions.**CTMDatasetNotFoundError**
Bases: LookupError

Missing CTM exception class.

8.1.7 pybdm.options module

Global package options.

pybdm.options.**warn_if_missing_ctm**
Should warnings for missing CTM values be sent.

Type bool

pybdm.options.**raise_if_zero**
Should error be raised in the case of zero BDM value, which is usually indicative of malformed data.

Type bool

pybdm.options.**get**(*name=None*)
Get option value or options dict.

Parameters **name** (*str or None*) – If None then the copy of the option dict is returned. If str then the given option value is returned.

See also:

set_options() description of the available global options

Raises **KeyError** – If *name* does not give a proper option name.

pybdm.options.**set**(*warn_if_missing_ctm=None*, *raise_if_zero=None*)
Set global package options.

Parameters

- **warn_if_missing_ctm** (*bool*) – Should warnings for missing CTM values be sent.
- **raise_if_zero** (*bool*) – Should error be raised in the case of zero BDM value, which is usually indicative of malformed data.

8.1.8 pybdm.partitions module

Partition algorithm classes.

Partition algorithms are used during the decomposition stage of BDM (see *Theory & Design* and *pybdm.bdm*), in which datasets are sliced into blocks of appropriate sizes.

Decomposition can be done in multiple ways that handles boundaries differently. This is why partition algorithms have to be properly configured, so it is well-specified what approach exactly is to be used.

class pybdm.partitions.**PartitionCorrelated**(*shape*, *shift=1*)

Bases: *pybdm.partitions.PartitionIgnore*

Partition with the 'correlated' boundary condition.

shape

Part shape.

Type tuple

shift

Shift parameter for the sliding window.

Type int (positive)

Notes

See *Theory & Design* for a detailed description.

Raises **AttributeError** – If *shift* is not positive.

decompose(*X*)

Decompose with the 'correlated' boundary.

_Partition.**decompose**(*X*)

Decompose a dataset into blocks.

Parameters **x** (*array_like*) – Dataset of arbitrary dimensionality represented as a *Numpy* array.

Yields *array_like* – Dataset blocks.

name = 'correlated'

property params

class pybdm.partitions.**PartitionIgnore**(*shape*)

Bases: pybdm.partitions._Partition

Partition with the 'ignore' boundary condition.

shape

Part shape.

Type tuple

Notes

See *Theory & Design* for a detailed description.

decompose(*X*)

Decompose with the 'ignore' boundary.

_Partition.**decompose**(*X*)

Decompose a dataset into blocks.

Parameters **x** (*array_like*) – Dataset of arbitrary dimensionality represented as a *Numpy* array.

Yields *array_like* – Dataset blocks.

name = 'ignore'

class pybdm.partitions.**PartitionRecursive**(*shape*, *min_length=2*)
Bases: pybdm.partitions._Partition

Partition with the 'recursive' boundary condition.

shape
Part shape.

Type tuple

min_length
Minimum parts' length. Non-negative. In case of multidimensional objects it specifies minimum length of any single dimension.

Type int

Notes

See *Theory & Design* for a detailed description.

decompose(*X*)
Decompose with the 'recursive' boundary.

_Partition.**decompose**(*X*)
Decompose a dataset into blocks.
Parameters **x** (*array_like*) – Dataset of arbitrary dimensionality represented as a *Numpy* array.
Yields *array_like* – Dataset blocks.

name = 'recursive'

property params

8.1.9 pybdm.utils module

Utility functions.

pybdm.utils.**decompose_dataset**(*X*, *shape*, *shift=0*)
Decompose a dataset into blocks.

Parameters

- **X** (*array_like*) – Daataset represented as a *Numpy* array.
- **shape** (*tuple*) – Slice shape.
- **shift** (*int*) – Shift value for slicing. Nonoverlaping slicing if non-positive.

Yields *array_like* – Dataset blocks.

Examples

```
>>> import numpy as np
>>> X = np.ones((5, 3), dtype=int)
>>> [ x for x in decompose_dataset(X, (3, 3)) ]
[array([[1, 1, 1],
       [1, 1, 1],
       [1, 1, 1]]), array([[1, 1, 1],
       [1, 1, 1]])]
```

pybdm.utils.**get_ctm_dataset**
Get CTM dataset by name.

This function uses a global cache, so each CTM dataset is loaded to the memory only once.

Parameters **name** (*str*) – Name of a dataset.

Returns CTM lookup table.

Return type dict

Raises **ValueError** – If non-existent CTM dataset is requested.

pybdm.utils.**iter_part_shapes**(*X*, *shape*, *shift=0*)
Iterate over part shapes induced by slicing.

Parameters

- **X** (*array_like*) – Dataset represented as a *Numpy* array.
- **shape** (*tuple*) – Slice shape.
- **shift** (*int*) – Shift value for slicing. Nonoverlaping slicing if non-positive.

Yields *tuple* – Part shapes.

Examples

```
>>> import numpy as np
>>> X = np.ones((5, 3), dtype=int)
>>> [ x for x in iter_part_shapes(X, (3, 3)) ]
[(3, 3), (2, 3)]
```

pybdm.utils.**iter_slices**(*X*, *shape*, *shift=0*)
Iter over slice indices of a dataset.

Slicing is done in a way that ensures that only pieces on boundaries of the sliced dataset can have leftovers in regard to a specified shape.

Parameters

- **X** (*array_like*) – Daataset represented as a *Numpy* array.
- **shape** (*tuple*) – Slice shape.
- **shift** (*int*) – Shift value for slicing. Nonoverlaping slicing if non-positive.

Yields *slice* – Slice indices.

Examples

```
>>> import numpy as np
>>> X = np.ones((5, 3), dtype=int)
>>> [ x for x in iter_slices(X, (3, 3)) ]
[(slice(0, 3, None), slice(0, 3, None)), (slice(3, 5, None), slice(0, 3, None))]
```

pybdm.utils.**list_ctm_datasets**()
Get a list of available precomputed CTM datasets.

Examples

```
>>> list_ctm_datasets()
['CTM-B2-D12', 'CTM-B2-D4x4', 'CTM-B4-D12', 'CTM-B5-D12', 'CTM-B6-D12', 'CTM-B9-
↪D12']
```

pybdm.utils.**prod**(*seq*)

Product of a sequence of numbers.

Parameters **seq** (*sequence*) – A sequence of numbers.

Returns Product of numbers.

Return type float or int

Notes

This is defined as:

$$\prod_{i=1}^{n} x_i$$

8.1.10 Module contents

PyBDM: Block Decomposition Method

This package provides the `pybdm.BDM` class for computing approximated algorithmic complexity of arbitrarily large binary 1D and 2D arrays as well as 1D arrays with 4, 5, 6 or 9 unique symbols based on the *Block Decomposition Method* (**BDM**). Theory and the design of the package are described in *Theory & Design*.

CHAPTER

NINE

PYBDM: PYTHON INTERFACE TO THE *BLOCK DECOMPOSITION METHOD*

The Block Decomposition Method (BDM) approximates algorithmic complexity of a dataset of arbitrary size, that is, the length of the shortest computer program that generates it. This is not trivial as algorithmic complexity is not a computable quantity in the general case and estimation of algorithmic complexity of a dataset can be very useful as it points to mechanistic connections between elements of a system, even such that do not yield any regular statistical patterns that can be captured with more traditional tools based on probability theory and information theory.

Currently 1D and 2D binary arrays are supported, as well as 1D arrays with 4, 5, 6 and 9 discrete symbols.

BDM and the necessary parts of the algorithmic information theory it is based on are described in [STZDG14] and [ZHOK+18].

9.1 Installation

Standard installation (stable):

```
pip install pybdm
```

Development version installation:

```
pip install git+https://github.com/sztal/pybdm.git
```

Local development:

```
git clone https://github.com/sztal/pybdm
cd pybdm
pip install --editable .
```

9.1.1 Supported versions

Python3.5+ is supported. Tests are run against Linux, but Windows and OSX should work as well.

9.2 Usage

The BDM is implemented using the object-oriented approach and expects input represented as Numpy arrays of integer type.

> `BDM` objects operate exclusively on **integer arrays**. Hence, any alphabet must be first mapped to a set of integers ranging from `0` to `k`. Currently only standard `numpy` arrays are accepted. However, in general it is possible to conceive of a BDM variant optimized for sparse array. We plan provide in the releas.

Detailed description of the design of our implementation of BDM can be found in *Theory & Design*.

9.2.1 Binary sequences (1D)

```
import numpy as np
from pybdm import BDM

# Create a dataset (must be of integer type)
X = np.ones((100,), dtype=int)

# Initialize BDM object
# ndim argument specifies dimensionality of BDM
bdm = BDM(ndim=1)

# Compute BDM
bdm.bdm(X)

# BDM objects may also compute standard Shannon entropy in base 2
bdm.ent(X)
```

9.2.2 Binary matrices (2D)

```
import numpy as np
from pybdm import BDM

# Create a dataset (must be of integer type)
X = np.ones((100, 100), dtype=int)

# Initialize BDM object
bdm = BDM(ndim=2)

# Compute BDM
bdm.bdm(X)

# BDM objects may also compute standard Shannon entropy in base 2
bdm.ent(X)
```

9.2.3 Non-binary sequences (1D)

```
import numpy as np
from pybdm import BDM

# Create a dataset (4 discrete symbols)
```

(continues on next page)

(continued from previous page)

```
np.random.seed(303)
X = np.random.randint(0, 4, (100,))

# Initialize BDM object with 4-symbols alphabet
bdm = BDM(ndim=1, nsymbols=4)

# Compute BDM
bdm.bdm(X)
```

9.2.4 Parallel processing

PyBDM was designed with parallel processing in mind. Using modern packages for parallelization such as joblib makes it really easy to compute BDM for massive objects.

In this example we will slice a 1000x1000 dataset into 200x200 pieces compute so-called counter objects (final BDM computation operates on such objects) in parallel in 4 independent processes, and aggregate the results into a single BDM approximation of the algorithmic complexity of the dataset.

> Remember that data has to be sliced correctly during parallelization in order to ensure fully correct BDM computations. That is, all slices except lower and right boundaries have to be decomposable without any boundary leftovers by the selected decomposition algorithm.

```
import numpy as np
from joblib import Parallel, delayed
from pybdm import BDM
from pybdm.utils import decompose_dataset

# Create a dataset (must be of integer type)
X = np.ones((1000, 1000), dtype=int)

# Initialize BDM object
bdm = BDM(ndim=2)

# Compute counter objects in parallel
counters = Parallel(n_jobs=4) \
    (delayed(bdm.decompose_and_count)(d) for d in decompose_dataset(X, (200, 200)))

# Compute BDM
bdm.compute_bdm(*counters)
```

9.2.5 Perturbation analysis

Besides the main *Block Decomposition Method* implementation *PyBDM* provides also an efficient algorithm for perturbation analysis based on *BDM* (or standard Shannon entropy).

A perturbation experiment studies change of *BDM* / entropy under changes applied to the underlying dataset. This is the main tool for detecting parts of a system having some causal significance as opposed to noise parts.

Parts which after yield negative contribution to the overall complexity after change are likely to be important for the system, since their removal make it more noisy. On the other hand parts that yield positive contribution to the overall complexity after change are likely to be noise since they extend the system's description length.

```
import numpy as np
from pybdm import BDM
```

(continues on next page)

(continued from previous page)

```
from pybdm.algorithms import PerturbationExperiment

# Create a dataset (must be of integer type)
X = np.ones((100, 100), dtype=int)

# Initialize BDM object
bdm = BDM(ndim=2)

# Initialize perturbation experiment object
# (may be run for both bdm or entropy)
perturbation = PerturbationExperiment(bdm, X, metric='bdm')

# Compute BDM change for all data points
delta_bdm = perturbation.run()

# Compute BDM change for selected data points and keep the changes while running
# One array provide indices of elements that are to be change.
idx = np.array([[0, 0], [10, 10]], dtype=int)
# Another array provide new values to assign.
# Negative values mean that new values will be selected
# randomly from the set of other possible values from the alphabet.
values = np.array([-1, -1], dtype=int)
delta_bdm = perturbation.run(idx, values, keep_changes=True)

# Here is an example applied to an adjacency matrix
# (only 1's are perturbed and switched to 0's)
# (so perturbations correspond to edge deletions)
X = np.random.randint(0, 2, (100, 100))
# Indices of nonzero entries in the matrix
idx = np.argwhere(X)
# PerturbationExperiment can be instantiated without passing data
pe = PerturbationExperiment(bdm, metric='bdm')
# data can be added later
pe.set_data(X)
# Run experiment and perturb edges
# No values argument is passed so perturbations automatically switch
# values to other values from the alphabet (in this case 1 --> 0)
delta_bdm = pe.run(idx)
```

9.3 Feedback

If you have any suggestions or questions about **PyBDM** feel free to email me at stalaga@protonmail.com.

If you encounter any errors or problems with **PyBDM**, please let me know! Open an Issue at the GitHub http://github.com/sztal/pybdm main repository.

BIBLIOGRAPHY

[STZDG14] Fernando Soler-Toscano, Hector Zenil, Jean-Paul Delahaye, and Nicolas Gauvrit. Calculating Kolmogorov Complexity from the Output Frequency Distributions of Small Turing Machines. *PLoS ONE*, 9(5):e96223, May 2014. URL: http://dx.plos.org/10.1371/journal.pone.0096223, doi:10.1371/journal.pone.0096223.

[ZHOK+18] Hector Zenil, Santiago Hernández-Orozco, Narsis Kiani, Fernando Soler-Toscano, Antonio Rueda-Toicen, and Jesper Tegnér. A Decomposition Method for Global Evaluation of Shannon Entropy and Local Estimations of Algorithmic Complexity. *Entropy*, 20(8):605, August 2018. URL: http://www.mdpi.com/1099-4300/20/8/605, doi:10.3390/e20080605.

[ZKZT19] Hector Zenil, Narsis A. Kiani, Allan A. Zea, and Jesper Tegnér. Causal deconvolution by algorithmic generative models. *Nature Machine Intelligence*, 1(1):58–66, January 2019. URL: http://www.nature.com/articles/s42256-018-0005-0, doi:10.1038/s42256-018-0005-0.

PYTHON MODULE INDEX

p

pybdm, 38
pybdm.algorithms, 23
pybdm.bdm, 25
pybdm.ctmdata, 23
pybdm.encoding, 31
pybdm.exceptions, 34
pybdm.options, 34
pybdm.partitions, 34
pybdm.utils, 36

INDEX

A

array_from_string() (*in module pybdm.encoding*), 31

B

BDM (*class in pybdm.bdm*), 26
bdm (*pybdm.algorithms.PerturbationExperiment attribute*), 24
bdm() (*pybdm.bdm.BDM method*), 27
BDMConfigurationError, 34
BDMRuntimeWarning, 34

C

compute_bdm() (*pybdm.bdm.BDM method*), 27
compute_ent() (*pybdm.bdm.BDM method*), 28
count() (*pybdm.bdm.BDM method*), 28
CTMDatasetNotFoundError, 34
ctmname (*pybdm.bdm.BDM attribute*), 26

D

decode_array() (*in module pybdm.encoding*), 31
decode_sequence() (*in module pybdm.encoding*), 31
decompose() (*pybdm.bdm.BDM method*), 28
decompose() (*pybdm.partitions.PartitionCorrelated method*), 35
decompose() (*pybdm.partitions.PartitionCorrelated._Partition method*), 35
decompose() (*pybdm.partitions.PartitionIgnore method*), 35
decompose() (*pybdm.partitions.PartitionIgnore._Partition method*), 35
decompose() (*pybdm.partitions.PartitionRecursive method*), 36
decompose() (*pybdm.partitions.PartitionRecursive._Partition method*), 36
decompose_and_count() (*pybdm.bdm.BDM method*), 29
decompose_dataset() (*in module pybdm.utils*), 36

E

encode_array() (*in module pybdm.encoding*), 32
encode_sequence() (*in module pybdm.encoding*), 32
ent() (*pybdm.bdm.BDM method*), 29

G

get() (*in module pybdm.options*), 34
get_ctm_dataset (*in module pybdm.utils*), 36

I

iter_part_shapes() (*in module pybdm.utils*), 37
iter_slices() (*in module pybdm.utils*), 37

L

list_ctm_datasets() (*in module pybdm.utils*), 37
lookup() (*pybdm.bdm.BDM method*), 30

M

metric (*pybdm.algorithms.PerturbationExperiment attribute*), 24
min_length (*pybdm.partitions.PartitionRecursive attribute*), 36

N

name (*pybdm.partitions.PartitionCorrelated attribute*), 35
name (*pybdm.partitions.PartitionIgnore attribute*), 35
nbdm (*pybdm.partitions.PartitionRecursive attribute*), 36
nbdm() (*pybdm.bdm.BDM method*), 30
ndim (*pybdm.bdm.BDM attribute*), 26
ndim() (*pybdm.algorithms.PerturbationExperiment property*), 24
nent() (*pybdm.bdm.BDM method*), 30
normalize_array() (*in module pybdm.encoding*), 33
normalize_key() (*in module pybdm.encoding*), 33
nsymbols (*pybdm.bdm.BDM attribute*), 26

P

params() (*pybdm.partitions.PartitionCorrelated property*), 35
params() (*pybdm.partitions.PartitionRecursive property*), 36

partition (*pybdm.bdm.BDM attribute*), 26
PartitionCorrelated (*class in pybdm.partitions*), 35
PartitionIgnore (*class in pybdm.partitions*), 35
PartitionRecursive (*class in pybdm.partitions*), 35
perturb() (*pybdm.algorithms.PerturbationExperiment method*), 24
PerturbationExperiment (*class in pybdm.algorithms*), 23
prod() (*in module pybdm.utils*), 38
pybdm (*module*), 38
pybdm.algorithms (*module*), 23
pybdm.bdm (*module*), 25
pybdm.ctmdata (*module*), 23
pybdm.encoding (*module*), 31
pybdm.exceptions (*module*), 34
pybdm.options (*module*), 34
pybdm.partitions (*module*), 34
pybdm.utils (*module*), 36

R

raise_if_zero (*in module pybdm.options*), 34
raise_if_zero (*pybdm.bdm.BDM attribute*), 26
run() (*pybdm.algorithms.PerturbationExperiment method*), 25

S

set() (*in module pybdm.options*), 34
set_data() (*pybdm.algorithms.PerturbationExperiment method*), 25
shape (*pybdm.partitions.PartitionCorrelated attribute*), 35
shape (*pybdm.partitions.PartitionIgnore attribute*), 35
shape (*pybdm.partitions.PartitionRecursive attribute*), 36
shape() (*pybdm.algorithms.PerturbationExperiment property*), 25
shift (*pybdm.partitions.PartitionCorrelated attribute*), 35
size() (*pybdm.algorithms.PerturbationExperiment property*), 25
string_from_array() (*in module pybdm.encoding*), 33

W

warn_if_missing_ctm (*in module pybdm.options*), 34
warn_if_missing_ctm (*pybdm.bdm.BDM attribute*), 26

X

X (*pybdm.algorithms.PerturbationExperiment attribute*), 24